GPU Performance Considerations

Dr. Juan Gómez Luna
Prof. Onur Mutlu

ETH Zürich
Spring 2023
7 April 2023
GPU Memories
Memory Hierarchy in CUDA Programs
Tiled Matrix-Matrix Multiplication (V)

__shared__ float A_s[TILE_DIM][TILE_DIM];
__shared__ float B_s[TILE_DIM][TILE_DIM];    --- Declare arrays in shared memory

unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;

float sum = 0.0f;

for(unsigned int tile = 0; tile < N/TILE_DIM; ++tile) {
    // Load tile to shared memory
    A_s[threadIdx.y][threadIdx.x] = A[row*N + tile*TILE_DIM + threadIdx.x];
    B_s[threadIdx.y][threadIdx.x] = B[(tile*TILE_DIM + threadIdx.y)*N + col];
    __syncthreads();    --- Threads wait for each other to finish loading before computing

    // Compute with tile
    for(unsigned int i = 0; i < TILE_DIM; ++i) {
        sum += A_s[threadIdx.y][i]*B_s[i][threadIdx.x];
    }
    __syncthreads();    --- Threads wait for each other to finish computing before loading
}

C[row*N + col] = sum;
Performance Considerations
Performance Considerations

- **Main bottlenecks**
  - CPU-GPU data transfers
  - Global memory access

- **Memory access**
  - Latency hiding
    - Occupancy
  - Memory coalescing
  - Data reuse
    - Shared memory usage

- **SIMD (Warp) Utilization:** Divergence

- **Other considerations**
  - Atomic operations: Serialization
  - Data transfers between CPU and GPU
    - Overlap of communication and computation
Memory Access
Latency Hiding via Warp-Level FGMT

- **Warp:** A set of threads that execute the same instruction (on different data elements)

- **Fine-grained multithreading**
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies

- **Register values of all threads stay in register file**

- **FGMT enables long latency tolerance**
  - Millions of pixels
Latency Hiding and Occupancy

- **FGMT** can hide **long latency operations** (e.g., memory accesses)
- **Occupancy**: ratio of **active warps** to the maximum number of warps per GPU core

![Diagram showing latency hiding and occupancy](image)
Occupancy

- GPU core, a.k.a. SM, resources (typical values)
  - Maximum number of warps per SM (64)
  - Maximum number of blocks per SM (32)
  - Register usage (256KB)
  - Shared memory usage (64KB)

- Occupancy calculation
  - Number of threads per block (defined by the programmer)
  - Registers per thread (known at compile time)
  - Shared memory per block (defined by the programmer)
CUDA Occupancy Calculator (III)

9. Occupancy Calculator

NVIDIA Nsight Compute provides an Occupancy Calculator that allows you to compute the multiprocessor occupancy of a GPU for a given CUDA kernel. It offers feature parity to the CUDA Occupancy Calculator spreadsheet.

The Occupancy Calculator can be opened directly from a profile report or as a new activity. The occupancy calculator data can be saved to a file using File > Save. By default, the file uses the .nCU-occ extension. The occupancy calculator file can be opened using File > Open File.

1. Launching from the Connection Dialog

Select the Occupancy Calculator activity from the connection dialog. You can optionally specify an occupancy calculator data file, which is used to initialize the calculator with the data from the saved file. Click the Launch button to open the Occupancy Calculator.

https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator
Memory Layout of a Matrix in C
DRAM Bank Operation

Access Address:
(Row 0, Column 0)
(Row 0, Column 1)
(Row 0, Column 85)
(Row 1, Column 0)

Row address 0

Row decoder

Columns

Row 1

Row Buffer

CONFLICT!

Column address 85

Column mux

Data
DRAM Burst

- Accessing data in different bursts (rows)
  - Need to access the array again

Timeline:

- Accessing data in the same burst (row)
  - No need to access the array again, just the multiplexerer

Timeline:

- Accessing data in the same burst is faster than accessing data in different bursts
Recall: Memory Banking

- Memory is divided into banks that can be accessed independently; banks share address and data buses (to minimize pin cost).
- Can start and complete one bank access per cycle.
- Can sustain N concurrent accesses if all N go to different banks.

![Memory Banking Diagram](credit: Derek Chiou)
Multiple Banks (Interleaving) and Channels

- Multiple banks
  - Enable concurrent DRAM accesses
  - Bits in address determine which bank an address resides in
- Multiple independent channels serve the same purpose
  - But they are even better because they have separate data buses
  - Increased bus bandwidth

- Enabling more concurrency requires reducing
  - Bank conflicts
  - Channel conflicts

- How to select/randomize bank/channel indices in address?
  - Lower order bits have more entropy
  - Randomizing hash functions (XOR of different address bits)
Latency Hiding with Multiple Banks

- With one bank, time still wasted in between bursts

- Latency can be hidden by having multiple banks

- Need many threads to simultaneously access memory to keep all banks busy
  - Achieved with having high occupancy in GPU cores (SMs)
    - Similar idea to hiding pipeline latency in the core
Memory Coalescing (I)

- When threads in the same warp access consecutive memory locations in the same burst, the accesses can be combined and served by one burst
  - One DRAM transaction is needed
  - Known as memory coalescing

- If threads in the same warp access locations not in the same burst, accesses cannot be combined
  - Multiple transactions are needed
  - Takes longer to service data to the warp
  - Sometimes called memory divergence

Slide credit: Izzat El Hajj
Memory Coalescing (II)

- When accessing global memory, we want to make sure that **concurrent threads access nearby memory locations**.
- **Peak bandwidth** utilization occurs when all threads in a warp access **one cache line** (or several consecutive cache lines).

---

---

Slide credit: Hwu & Kirk
Uncoalesced Memory Accesses

Access direction in Kernel code

Time Period 1

M₀₀ M₁₀ M₂₀ M₃₀
M₀₁ M₁₁ M₂₁ M₃₁
M₀₂ M₁₂ M₂₂ M₃₂
M₀₃ M₁₃ M₂₃ M₃₃

Time Period 2

T₁ T₂ T₃ T₄
M₀₀ M₁₀ M₂₀ M₃₀
M₀₁ M₁₁ M₂₁ M₃₁
M₀₂ M₁₂ M₂₂ M₃₂
M₀₃ M₁₃ M₂₃ M₃₃

...
Coalesced Memory Accesses

Access direction in Kernel code

Time Period 1
\[ T_1 \quad T_2 \quad T_3 \quad T_4 \]

Time Period 2
\[ T_1 \quad T_2 \quad T_3 \quad T_4 \]

\[ \cdots \]

Slide credit: Hwu & Kirk
Use Shared Memory to Improve Coalescing

Original Access Pattern

Tiled Access Pattern

Copy into scratchpad memory

Perform multiplication with scratchpad values

Slide credit: Hwu & Kirk
Data Reuse

- Same memory locations accessed by neighboring threads

```c
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```
To take advantage of data reuse, we divide the input into tiles that can be loaded into shared memory.

```c
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
...
Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
    }
}
```
Shared Memory

- Shared memory is an **interleaved (banked) memory**
  - Each bank can service one address per cycle

- Typically, 32 banks in NVIDIA GPUs
  - Successive 32-bit words are assigned to successive banks
    - Bank = Address % 32

- Bank conflicts are **only possible within a warp**
  - No bank conflicts between different warps
Shared Memory Bank Conflicts (I)

- **Bank conflict free**

  ![Diagram showing bank conflict free addressing]

  Linear addressing: stride = 1

  Random addressing 1:1

*Slide credit: Hwu & Kirk*
Shared Memory Bank Conflicts (II)

- N-way bank conflicts

2-way bank conflict: stride = 2

8-way bank conflict: stride = 8

Slide credit: Hwu & Kirk
Reducing Shared Memory Bank Conflicts

- Bank conflicts are only possible within a warp
  - No bank conflicts between different warps

- If strided accesses are needed, some optimization techniques can help
  - Padding
  - Randomized mapping
  - Hash functions
SIMD Utilization
Intra-warp divergence

Compute(threadIdx.x);
if (threadIdx.x % 2 == 0){
   Do_this(threadIdx.x);
}
else{
   Do_that(threadIdx.x);
}
Increasing SIMD Utilization

- **Divergence-free execution**

```c
Compute(threadIdx.x);
if (threadIdx.x < 32) {
    Do_this(threadIdx.x * 2);
}
else {
    Do_that((threadIdx.x%32)*2+1);
}
```
Vector Reduction: Naïve Mapping (I)

0+1 2+3 4+5 6+7 8+9 10+11

0...3 4..7 8..11

0..7 8..15

Thread 0 Thread 2 Thread 4 Thread 6 Thread 8 Thread 10

Thread 0
Thread 2
Thread 4
Thread 6
Thread 8
Thread 10

0...3
4..7
8..11

0..7
8..15

Slide credit: Hwu & Kirk
Divergence-Free Mapping (I)

- All active threads belong to the same warp

![Diagram showing the concept of Divergence-Free Mapping (I)]
Atomic Operations
Atomic Operations (I)

- CUDA provides **atomic instructions** on shared memory and global memory
  - They perform **read-modify-write** operations atomically

- Arithmetic functions
  - Add, sub, max, min, exch, inc, dec, CAS
    ```c
    int atomicAdd(int*, int);
    ```
  - Pointer to shared memory or global memory
  - Value to add
  - Return value (old value)

- Bitwise functions
  - And, or, xor

- Datatypes: int, uint, ull, float (half, single, double)*

* Datatypes for different atomic operations in [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions)
Atomic Operations (II)

- Atomic operations serialize the execution if there are atomic conflicts.

No atomic conflict = concurrent updates

Atomic conflict = serialized updates
Uses of Atomic Operations

- **Computation**
  - Atomics on an array that will be the output of the kernel
  - Example
    - Histogram, reduction

- **Synchronization**
  - Atomics on memory locations that are used for synchronization or coordination
  - Example
    - Counters, locks, flags...

- Use them to prevent **data races** when more than one thread need to update the same memory location
Asynchronous Data Transfers between CPU and GPU
CUDA Streams

- **CUDA streams** (command queues in OpenCL)
- **Sequence of operations that are performed in order**
  - 1. Data transfer CPU-GPU
  - 2. Kernel execution
    - D input data instances, B blocks
    - #Streams: \((D / \#\text{Streams})\) data instances, \((B / \#\text{Streams})\) blocks
  - 3. Data transfer GPU-CPU
Asynchronous Transfers between CPU & GPU

- **Computation** divided into #Streams
  - D input data instances, B blocks
  - #Streams
    - D/#Streams data instances
    - B/#Streams blocks

- Estimates
  - \( t_E + \frac{t_T}{\#Streams} \)  \( t_E \geq t_T \) (dominant kernel)
  - \( t_T + \frac{t_E}{\#Streams} \)  \( t_T > t_E \) (dominant transfers)
Applications with independent computation on different data instances can benefit from asynchronous transfers. For instance, video processing.
Performance Considerations

- Main bottlenecks
  - CPU-GPU data transfers
  - Global memory access

- Memory access
  - Latency hiding
    - Occupancy
  - Memory coalescing
  - Data reuse
    - Shared memory usage

- SIMD (Warp) Utilization: Divergence

- Other considerations
  - Atomic operations: Serialization
  - Data transfers between CPU and GPU
    - Overlap of communication and computation
Recommended Readings (II)

  - Chapter 6 - Performance considerations
  - Chapter 20 - Programming a heterogeneous computing cluster, Section 20.5
AoS vs. SoA

- **Array of Structures vs. Structure of Arrays**

```c
struct foo{
    float a[8];
    float b[8];
    float c[8];
    int d[8];
} A;
```

```c
struct foo{
    float a;
    float b;
    float c;
    int d;
} A[8];
```

---

**HetSys Course: Lecture 5: GPU Performance Considerations (Fall 2022)**

- Onur Mutlu Lectures

711 views 5 months ago  Livestream - P&S Programming Heterogeneous Computing Systems with GPUs and other Accelerators (Fall 2022)

Project & Seminar, ETH Zürich, Fall 2022

Programming Heterogeneous Computing Systems with GPUs and other Accelerators ([https://safari.ethz.ch/projects_and_s...](https://safari.ethz.ch/projects_and_s...))

Show more

P&S Heterogeneous Systems

GPU Performance Considerations

Dr. Juan Gómez Luna
Prof. Onur Mutlu

ETH Zürich
Spring 2023
7 April 2023